Assignment 1 solution

The assignment has the following requirements

- The program must implement a 0-7 counter
- The current count must be represented by a color on the on-board RGB LED
  - 0: Off
  - 1: Green
  - 2: Blue
  - 3: Cyan
  - 4: Red
  - 5: Yellow
  - 6: Magenta
  - 7: White
- The counter must be able to count up and down
- The counter should proceed on step with a button push
- The direction of the counter must change with a double button push
- The counter must enter AUTO mode when the button is held for 2 seconds or more
  - When in AUTO mode the counter must proceed one step every 200 ms
- A button push while in AUTO mode should return the counter to manual mode

The full assignment can be read in Assignment1.pdf.

This article covers the buildup of the system starting with the basics and building on top of that to lastly implement all the criteria from above. For the full code, see my GitHub.

Topics covered

- Interrupts (GPIO Interrupts)
- Counters/Timers (SysTick, Systick Interrupt)

Basics

I'll start with the basics by implementing basic logic to change through the colors and enabling the pins correctly.

Criteria covered in this section:

- The program must implement a 0-7 counter
- The current count must be represented by a color on the on-board RGB LED

    - 0: Off
    - 1: Green
    - 2: Blue
    - 3: Cyan
    - 4: Red
    - 5: Yellow
    - 6: Magenta
    - 7: White
- The counter must be able to count up ~~and down~~
- The counter should proceed on step with a button push

Colors

The RBG LED which will be used for this can be found on port F of the GPIO. The LED pins are as follows

- Red: PF1
- Blue: PF2
- Green: PF3

We can use this to cycle through the colors by enabling the RBG pins, which will combine to the colors.
The colors, the corresponding count, and their corresponding HIGH/LOW pins can be seen in the following table.

Counter Value	PF1 (red)	PF2 (blue)	PF3 (green)	Color
0	0	0	0	Off
1	0	0	1	Green
2	0	1	0	Blue
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Yellow
6	1	1	0	Magenta
7	1	1	1	White

When implementing this in code, it is important to remember, that we will be setting the values for *all* Port F pins (bit 0-7). Since we only want to set PF1 - PF3, we will leave the rest as zero.

```
// Initialize colors {PF1, PF2, PF3} = {r, b, g}
const int off = 0x00; // 0000 0000
```

```
const int green = 0x08; // 0000 1000
const int blue = 0x04; // 0000 0100
const int cyan = 0x0C; // 0000 1100
const int red = 0x02; // 0000 0010
const int yellow = 0x0A; // 0000 1010
const int magenta = 0x06; // 0000 0110
const int white = 0x0E; // 0000 1110

// Color pointer vector to point to the colors
int colors[8] = {off, green, blue, cyan, red, yellow, magenta, white};

// Current count = current color
int cnt = 0;

// To be used in later development
bool dirUp = true;
bool autoMode = false;
```

We will use this to refer to later, to set the color on the LED.

Pins

Next we will enable the pins

We will start by enabling power and clock control to Port F, which will be the one we are using.

Next a dummy is used to implement a few dummy cycles in order to give Port F time to initialize.

The *direction* of the Port F pins are set as either output or input depending on their bit value.

A pull-up resistor is enabled for the switch, meaning it will be HIGH when not pressed an LOW when pressed.

Lastly digital function is enabled for Port F 1 - Port F 4 (PF1 - PF4) meaning we can actually code with them.

```
int dummy; // Dummy to do a few cycles

// Enable GPIO port F (used for RBG and switch) by turning on power to the
port
SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF; // System Control Run-Mode Clock
Gating Control

// Do a dummy read to insert a few cycles after enabling the peripheral
dummy = SYSCTL_RCGC2_R;
```

```
// Enable RGB (PF1 - PF3) as output and switch (PF4) as input
GPIO_PORTF_DIR_R = 0x0E; // 0000 1110 - Bit 1 is output, bit 0 is input

// Enable pull-up resister for switch (PF4)
GPIO_PORTF_PUR_R = 0x10; // 0001 0000

// Enable digital function for PF1 - PF4
GPIO_PORTF_DEN_R = 0x1E;
```

While loop

Lastly, we implement an infinite while loop, that reads the value of PF4 and increments cnt if PF4 is pressed.

It will also display the current count on the RBG LED

```
// Loop forever
while(1){
// Check if switch is pressed (active LOW with pull-up)
    if (!(GPIO_PORTF_DATA_R & 0x10)) {
        cnt = (cnt + 1) % 8; // Cycle through 0-7

    // Wait for button release as to not increment indefinitely
    while(!(GPIO_PORTF_DATA_R & 0x10));
    }

    // Set LED color (clear LED bits and set new color)
    GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | colors[cnt];
}
```

If statement

We check if the data register from Port F has the switch enabled. This done by performing an AND operation on PORTF*DATA and 0001 0000, where the _1* represents PF4 (being the 5th bit from the right (0, 1, 2, 3, 4)).

If the button is pressed, cnt is incremented, and we perform modulus division on it to keep it within 0-7

A while loop is implemented again, which just runs and blocks indefinitely while the button is pressed.

Setting LED

The LED is set from the colors defined before.

- We start by *clearing* the LEDs by performing an AND operation on PORTF_DATA and `~0x0E` which is 0000 1110 inverted (so 1111 0001). By doing this, we turn off PF1

- PF3.
- Next we perform an OR operation on this, and the color of our choice (decided by the count).
- This effectively sets the bits of the PF1 - PF3 to the ones of the color.

ISR

We now want to move the button logic to an Interrupt Service Routine (ISR).
This requires a few steps and changing in the startup file for the specific chip (TM4C123GH6PM). This is provided by the CCS program.

Criteria covered in this section: The same criteria as the Basics, but now with smarter logic.

Setting switch

We want to activate the ISR at a falling edge of our switch (being pushed).

```
// Set switch (PF4) as edge-sensitive
GPIO_PORTF_IS_R = 0x00;

// Trigger controlled by IEV
GPIO_PORTF_IBE_R = 0x00;

// Falling Edge Trigger
GPIO_PORTF_IEV_R = 0x00;

// Clear any Prior Interrupts
GPIO_PORTF_ICR_R = 0x10;

// Unmask interrupts for PF4
GPIO_PORTF_IM_R = 0x10;
```

By setting the IS_R value for PF4 to 0 (including the rest, as they are not used for this), we set it to edge-sensitive instead of level-sensitive (HIGH/LOW).

We next set *IBE* of PF4 to 0, meaning that we disable *Interrupt Both Edges*.

Instead we enable *IEV* on falling edge by setting the value for PF4 to 0.

- If we set it to 1, we would be detecting on *rising edge*.

We clear any pending interrupts on PF4 *before* enabling the pin for Interrupts. This is done by setting *ICR* of PF4 to 1.

Lastly, PF4 is enabled - or *unmasked* - by setting the *IM* value of PF4 to 1 (enable).

Enable interrupts from Port F in NVIC

To actually *use* the interrupts from port F, we will need to enable interrupt signals from this in the NVIC (Nested Vector Interrupt Controller).

NVIC is used by the ARM Cortex-M4 for interrupt management hardware. It manages the following:

- Prioritizes and manages all peripheral interrupts
- Automatically saves/restores processor context when entering/leaving an ISR
- Supports *nested interrupts* (higher priority can preempt lower priority)

```
NVIC_EN0_R |= (1 << (INT_GPIOF - INT_GPIOA));
```

We enable interrupts in NVIC from port F from this simple line.

`NVIC_EN0_R` is a 32-bit (0 - 31) hardware register that enables interrupts in the processor. By setting a bit to *1* we enable that interrupt.

Interrupt assignments start from *16* with `INT_GPIOA`. If we want to find `INT_GPIOF`'s placement in NVIC, we will need to subtract `INT_GPIOA` from the placement of `INT_GPIOF`. Since GPIO Port A is *16* and GPIO Port F is *46* the result will be *30*.

Since we want to now enable NVIC *30* we will need to shift bit 1 *left* by 30 bits.

- Remember LSB is to the right, and MSB is to the left.

We now perform an OR operation on `NVIC_EN0_R` to *enable* bit 30, and activate Port F.

- Sidenote: If a bit *higher* than 31 needs to be activated, `NVIC_EN1_R` can be used and so on for higher bits.
  - `NVIC_EN1_R`: 32-63
  - `NVIC_EN2_R`: 64-95
  - And so on.

Interrupt Service Routine (ISR)

Now that we have set up alle the basics for enabling ISR, we will need to define *what to do*.

Before defining our ISR, we will need to let the system know, that it should be used for interrupts from port F.

For this, we navigate to `tm4c123gh6pm_startup_ccs.c` which is a startup file created by Code Composer Studio (CCS).

```
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
```

```
static void IntDefaultHandler(void);

// Custom ISR handler
void GPIOF_Handler(void);
```

We can see her the *declaration* of a bunch of default fault handlers.

- We add the declaration of our custom interrupt handler to this.
- We will NOT define it just yet.

We next need to let the system know to use this on port F.
We navigate further down the file to find the *vector table* that defines what interrupts from parts of the system will use to handle the interrupt.

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),

    // The initial stack pointer

    ResetISR, // The reset handler

    NmiSR, // The NMI handler

    FaultISR, // The hard fault handler

    IntDefaultHandler, // The MPU fault handler

    IntDefaultHandler, // The bus fault handler

    IntDefaultHandler, // The usage fault handler

    0, // Reserved

    // ...

    IntDefaultHandler, // System Control (PLL, OSC, BO)
    IntDefaultHandler, // FLASH Control

    // Custom handler
    GPIOF_Handler, // GPIO Port F

    IntDefaultHandler, // GPIO Port G
    IntDefaultHandler, // GPIO Port H

    // ...
```

```
};
```

We can see a section of the vector here with examples of how it is defined.

As we can see, a lot of the interrupts, including the Ports use the default interrupt handler, `IntDefaultHandler`.

- We change the handler of GPIO Port F to our custom handler, `GPIOF_Handler`.

We now go back to `main.c` to *define* our ISR

```
// The Interrupt Service Routine (ISR)
void GPIOF_Handler(void) {

    // For debounce
    volatile int i;

    // Clear the interrupt flag for PF4 (must be done first)
    GPIO_PORTF_ICR_R = 0x10;

    // Increment counter to cycle through colors
    cnt = (cnt + 1) % 8; // Cycle through 0-7

    // Simple debounce delay
    for(i = 0; i < 100000; i++);

    // Wait for button release as to not increment indefinitely
    while(!(GPIO_PORTF_DATA_R & 0x10));
}
```

This is relatively simple. We first define a volatile integer `i` to be used for a simple debounce delay later on.

The interrupt flag for PF4 will need to be done first, as to make room for future interrupts.

Next we increment cnt. We will however need to keep it within 0-7. This is simply done be performing *modulus* division. The clever reader will notice that this step is the same as when we covered the basics.

Lastly we implement the debounce delay by blocking the processor for a short time by increment `i` from 0 to 100.000.

To add to this, we also wait for the button to be released. This is not essential, and will be removed later on to make room for further functionality.

New while loop

The while loop from the Basics can now be shortened to only include LED control

```
// Loop forever
while(1) {
    // Set LED color (clear LED bits and set new color)
    GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | colors[cnt];
}
```

And the system will now react as before, but now with the button logic handled by an Interrupt Service Routine.

SysTick and automode

Next we want to implement SysTick as to effectively increment the count every 200 ms.

Criteria covered in this section:

- The counter must enter AUTO mode ~~when the button is held for 2 seconds or more~~
  - When in AUTO mode the counter must proceed one step every 200 ms

Enabling SysTick

We first want to enable SysTick to actually use it. We start by defining a couple of things, we will need later on, including somethings not defined in the header file/data sheet for the processor.

```
#define SYSTICK_RELOAD_VALUE 3199999 // 200 ms delay (16 MHz / 5 - 1 =
3,199,999 cycles)

// Missing definitions in tm4c123gh6pm.h file
#define NVIC_INT_CTRL_PEND_SYST 0x04000000 // Pend a systick int
#define NVIC_INT_CTRL_UNPEND_SYST 0x02000000 // Unpend a systick int

#define SYSTICK_PRIORITY 0x7E
```

We can see that we have defined the reload value for our timer. This is calculated from our desired millisecond delay of 200 ms and the clock period (16 MHz for this processor).

$$\text{Reload} = \frac{0.2\ \text{s}}{1/16\ \text{MHz}} - 1 = 3,199,999$$

Now we will setup and configure SysTick to suit our needs.

```
// Disable systick timer
NVIC_ST_CTRL_R &= ~(NVIC_ST_CTRL_ENABLE);
```

```
// Set current timer to reload value
NVIC_ST_CURRENT_R = SYSTICK_RELOAD_VALUE;
// Set Reload valye in systick reload register
NVIC_ST_RELOAD_R = SYSTICK_RELOAD_VALUE;

// NVIC systick setup, vector number 15 in startup_css.c
// Clear pending systick interrupt request
NVIC_INT_CTRL_R |= NVIC_INT_CTRL_UNPEND_SYST;

// Set systick priority to 0x10. First clear, then set
NVIC_SYS_PRI3_R &= ~(NVIC_SYS_PRI3_TICK_M);
NVIC_SYS_PRI3_R |= (NVIC_SYS_PRI3_TICK_M &
(SYSTICK_PRIORITY<<NVIC_SYS_PRI3_TICK_S));

// Select systick clock source, use core clock
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_CLK_SRC;

// Enable systick interrupt
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_INTEN;

// ...

// Enable and start systick timer
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_ENABLE;
```

This is done according to the explanation on SysTick.

SysTick CTRL

We can see the SysTick control register here. This is used to configure the clock for the SysTick timer, enable counter, enable SysTick interrupt and provide status of the counter.

```
// Disable systick timer
NVIC_ST_CTRL_R &= ~(NVIC_ST_CTRL_ENABLE);

// ...

// Select systick clock source, use core clock
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_CLK_SRC;

// Enable systick interrupt
```

```
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_INTEN;

// ...

// Enable and start systick timer
NVIC_ST_CTRL_R |= NVIC_ST_CTRL_ENABLE;
```

We will start by *disabling* the SysTick timer so it will not run in the background while we set it up. This is done by setting the enable bit to *0*.

We then do some other configuration, which will be covered shortly.

When we next use the SysTick CTRL register, we set it to use the core clock as its source. This is done by setting the `CLK_SRC` bit in the register to *1*.

We then enable SysTick interrupt, so that we can actually use it to interrupt the code and change some values. This is done by setting the `INTEN` bit of the register to *1*.

Lastly, when all other code is setup, and we are ready to run, we enable SysTick again, by setting the `ENB` (enable bit) to *1* again.

SysTick Reload

We can see on line 3, that we use `NVIC_ST_RELOAD_R` to set our calculated reload value into the SysTick register.

This will be copied to the counter on every SysTick interrupt. It basically holds our desired interrupt interval.

```
// ...

// Set Reload valye in systick reload register
NVIC_ST_RELOAD_R = SYSTICK_RELOAD_VALUE;

// ...
```

SysTick Current

We also set `NVIC_ST_CURRENT_R` on line 2. This basically loads the reload value in to the current value of SysTick, which will begin counting down to 0, when we enable SysTick

```
// ...
```

```
// Set current timer to reload value
NVIC_ST_CURRENT_R = SYSTICK_RELOAD_VALUE;

// ...
```

NVIC configuration

We will need to configure NVIC as to actually use SysTick as an interrupt.

```
// ...

// NVIC systick setup, vector number 15 in startup_css.c
// Clear pending systick interrupt request
NVIC_INT_CTRL_R |= NVIC_INT_CTRL_UNPEND_SYST;

// Set systick priority to 0x10. First clear, then set
NVIC_SYS_PRI3_R &= ~(NVIC_SYS_PRI3_TICK_M);
NVIC_SYS_PRI3_R |= (NVIC_SYS_PRI3_TICK_M &
(SYSTICK_PRIORITY<<NVIC_SYS_PRI3_TICK_S));

// ...
```

We will firstly need to clear pending SysTick interrupts. This is done on line 4, using the `NVIC_INT_CTRL_UNPEND_SYST` we defined before.

Secondly we set the priority of SysTick to 16, by first clearing it and then setting it.

- Clearing is simply done by inverting the SysTick Exception Priority and removing that sequence from `NVIC_SYS_PRI3_R`.

SysTick Interrupt

Since we enabled the SysTick interrupt bit in SysTick CTRL we will need to implement an ISR. This is done in much the same way as when we implemented it for the GPIO pin before.

We navigate to `tm4c123gh6pm_startup_ccs.c` to declare our custom handler.

```
void ResetISR(void);
static void NmiSR(void);
static void FaultISR(void);
static void IntDefaultHandler(void);

// Custom ISR handler
void GPIOF_Handler(void);
void SysTick_Handler(void);
```

We now need to let the system know, to use this on SysTick. We again navigate to the vector table further down the startup file.

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),

    // The initial stack pointer

    ResetISR, // The reset handler

    NmiSR, // The NMI handler

    FaultISR, // The hard fault handler

    IntDefaultHandler, // The MPU fault handler

    IntDefaultHandler, // The bus fault handler

    IntDefaultHandler, // The usage fault handler

    0, // Reserved

    // ...

    0, // Reserved
    IntDefaultHandler, // The PendSV handler

    // Custom Handler
    SysTick_Handler, // The SysTick handler

    IntDefaultHandler, // GPIO Port A
    IntDefaultHandler, // GPIO Port B

    // ...

};
```

And we then go back into `main.c` to define our ISR

This is a simple counter for now, but will be expanded upon, when we add the button functionality later.

```
// SysTick ISR
void SysTick_Handler(void) {
    if (dirUp) cnt = (cnt + 1) % 8; // Increment within 0-7
```

```
    else cnt = (cnt + 7) % 8; // Decrement within 0-7
}
```

While loop

The while loop remains the same as before.

```
// Loop forever
while(1) {
    // Set LED color (clear LED bits and set new color)
    GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | colors[cnt];
}
```

Now the button will still increment if pressed, but the timed counter runs automatically on startup. *We will need to change this*.

Expanded button functionality

Now that all other criteria are covered, we will need to implement the last button functionalities.

Criteria covered in this section

- The counter must be able to count up and down
- The counter should proceed on step with a button push
- The direction of the counter must change with a double button push
- The counter must enter AUTO mode when the button is held for 2 seconds or more
    - When in AUTO mode the counter must proceed one step every 200 ms
- A button push while in AUTO mode should return the counter to manual mode

We will not be changing the `main` function, only the previously covered ISRs.

Global values

To track the state of the button and properly act upon it, we introduce a few new global values to the program

```
#define DOUBLE_CLICK_WINDOW 300  // ms window for double-click detection
#define HOLD_THRESHOLD 2000      // ms for hold detection

// Millisecond counter (unsigned to handle overflow properly)
volatile unsigned int milliseconds = 0;

// Button state tracking
volatile unsigned int lastPressTime = 0;
volatile bool buttonPressed = false;
```

```
volatile unsigned int buttonPressStartTime = 0;
volatile bool holdDetected = false;
volatile bool waitingForSecondClick = false;
```

We see that the new `int` variables also have the type `unsigned`. This is done to avoid overflow, as we only work with 32 bit values (since that is what the registers of the Tiva C Series can hold), and milliseconds may quickly run over this limit.

- In reality this is not needed for this specific program/use case, as it would take around $2^{32}\ \text{ms} \approx 50\ \text{days}$ to overflow. This is well over the expected limit for the program, but I wanted to play around with it.

We implement these new variables to keep track of the time *between* button-presses (to check for double-click) and to check if the button is currently pressed or being held (to check for holding action).

New values are also defined at the top as limits for either double click or button hold millisecond delays.

The new values are implemented in the ISRs following this section.

Button ISR

```
// The Interrupt Service Routine (ISR)
void GPIOF_Handler(void) {
    // For debounce
    volatile int i;

    // Clear the interrupt flag for PF4 (must be done first)
    GPIO_PORTF_ICR_R = 0x10;

    // Simple debounce delay
    for(i = 0; i < 100000; i++);

    // If in auto mode, any button press exits auto mode
    if (autoMode) {
        autoMode = false;
        return;
    }

    unsigned int currentTime = milliseconds;
    unsigned int timeDiff = currentTime - lastPressTime;

    // Check for double-click
    if (waitingForSecondClick && timeDiff < DOUBLE_CLICK_WINDOW) {
        // Double-click detected! Toggle direction
        dirUp = !dirUp;
        waitingForSecondClick = false;
```

```
            holdDetected = true;  // Prevent hold and single-click actions
        } else {
            // Potential single click or hold - start tracking
            waitingForSecondClick = true;
            buttonPressed = true;
            buttonPressStartTime = currentTime;
            holdDetected = false;
        }

        lastPressTime = currentTime;
    }
```

We can see that the start of the button ISR is much the same as it was when we first implemented it before, with the debounce and clearing the interrupt flag.

Now however we also have a quick check to exit `automode` if any button press occurs while in `automode`.

We see next that we calculate the time between the current button press and the last button press. This is used in the `if` statement to check for a possible double click if we are currently waiting for a second click and if we are within the time limit set before of 300 milliseconds.

- If a double click was performed, the program changes the direction of counting.

If we do not detect any double click, it must have been either a single click or hold, and we start tracking this.

- The action of our click being time-dependent is now tracked in the SysTick ISR.

Last but not least, the last button press time must be the current time, as we pressed the button.

SysTick ISR

Next we explore the new ISR for SysTick

```
// SysTick ISR - runs every 200ms
void SysTick_Handler(void) {
    // Update millisecond counter (200ms per tick)
    milliseconds += 200;

    // Check for hold detection
    if (buttonPressed && !holdDetected) {
        // Check if button is still held down
        if (!(GPIO_PORTF_DATA_R & 0x10)) {
            unsigned int holdDuration = milliseconds -
buttonPressStartTime;
            if (holdDuration >= HOLD_THRESHOLD) {
```

```
                // Hold detected! Toggle auto mode
                autoMode = !autoMode;
                holdDetected = true;
                waitingForSecondClick = false;  // Cancel any pending
double-click
            }
        } else {
            // Button was released before hold threshold
            buttonPressed = false;
        }
    }

    // Auto mode - cycle through colors
    if (autoMode) {
        if (dirUp) cnt = (cnt + 1) % 8; // Increment within 0-7
        else cnt = (cnt + 7) % 8; // Decrement within 0-7
    }

    // Reset waiting flag after window expires
    if (waitingForSecondClick && (milliseconds - lastPressTime) >=
DOUBLE_CLICK_WINDOW) {
        waitingForSecondClick = false;
        // Process single click if not hold
        if (!holdDetected) {
            // Single click: increment or decrement based on dirUp
            if (dirUp) cnt = (cnt + 1) % 8;
            else cnt = (cnt + 7) % 8;
        }
    }
}
```

We see first that we increment the global milliseconds value by 200 milliseconds, since the ISR for SysTick runs every 200 ms

- In hindsight, a smaller value could have been chosen to speed up the reaction time of the system, but 200 ms still works properly, and frees up the CPU by not running as often.

We first check if the button has been flagged as being pressed. This activates the check for hold detection.

- The program first checks if the button is still being held down (since last time the ISR ran).
  - If so, we must also check if we have held the button for 2000 milliseconds (2 seconds) or more yet.
  - If the button has been held for 2 or more seconds, automode is activated and the program now counts on its own.

- If the button is not still being held, it was released before the threshold, and thus we do nothing yet.

If the program enters automode, it will start counting either up or down based on the chosen direction within a 0-7 count frame.

Lastly, if the window for double click (300 milliseconds) expires, we are no longer waiting for a double click, and instead either a hold action is coming up, or we pressed the button a single time.

- The difference between this is checked by checking if a hold has currently been detected. Since `holdDetected` is only true while inside the `if` statement regarding hold previously discussed, it will be false for a single click.

If a single click is detected, the program counts either up or down by one value within a 0-7 count frame.

This concludes the program and its explanation. Remember, if you want to check out the full code, checkout my GitHub.